Studying the Behaviour of the Single Parent Rule in Real-Time Java[*]

M. Teresa Higuera-Toledano

Facultad Informática, Universidad Complutense de Madrid, Ciudad Universitaria, 28040 Madrid Spain
mthiguer@dacya.ucm.es

Abstract. This paper addresses the issue of improving the performance of memory management for real-time Java applications, building upon the Real-Time Specification for Java (RTSJ). This specification imposes strict assignment rules to or from memory areas preventing the creation of dangling pointers, and thus maintaining the pointer safety of Java. The dynamic issues that Java presents, requires for some cases ensuring the checking of these rules at run-time, which adversely affects both the performance and predictability of the RTSJ application. This paper presents an efficient algorithm for managing scoped areas which requires some modifications in the current RTSJ specification.

1 Introduction

The *Real-time Specification for Java* (RTSJ) [6] introduces the concept of scoped memory to Java by extending the Java memory model to provide several kinds of memory areas: the *garbage-collected heap*, *immortal memory* areas that are never garbage collected, and *scoped memory* areas that are collected when there is not a thread using the memory area. The lifetime of objects allocated in scoped areas is governed by the control flow. Because scoped areas can be reclaimed at any time, objects within an area with a longer lifetime are not allowed to create a reference to an object within another area with a potentially shorter lifetime. Strict assignment rules placed on assignments to or from memory areas prevent the creation of dangling pointers. An RTSJ implementation must enforce these scope checks before executing an assignment.

This paper focuses on data structure and algorithms supporting the RTSJ assignment rules. In order to enforce the safety assignment rules in a more efficient way than the implementation proposed in [3], we study the behaviour of the single parent rule, and propose an alternative implementation based on the current edition of RTSJ [6]. We present an in depth description of the RTSJ memory model (Section 2). We

[*] Founded by the Ministerio de Ciencia y Tecnología of Spain (CICYT); Grant Number TIC2003-1321.

suggest a solution to improve the RTSJ memory model, which can be implemented efficiently by using simple data structures and algorithms (Section 3). We provide an outline of related work (Section 4). Finally a summary of our contribution conclude this paper (Section 5).

2 The Scoped Memory Model Behavior

RTSJ makes distinction between tree main kinds of tasks: *(i) low-priority* that are tolerant with the GC, *(ii) high-priority* that cannot tolerate unbounded preemption latencies, and *(iii)* c*ritical* that cannot tolerate preemption latencies. Since immortal and scoped areas are not garbage collected, they may be exploited by critical tasks. Several related threads, possibly real-time, can share a memory area, and the area must be active until at least the last thread has exited. The way that threads access objects within memory areas in the current RTSJ edition is governed by the following rules:

1. A traditional thread can allocate memory only on the traditional heap.
2. High-priority tasks may allocate memory from the heap, or from a memory area other than the heap by making that area the current allocation context.
3. Critical tasks must allocate memory from a memory area other than the heap by making that area the current allocation context.
4. A new allocation context is entered by calling the `enter()` method or by starting a real-time thread (i.e. a task or an event handler). Once an area is entered, all subsequent uses of the new keyword, within the program logic, will allocate objects from the memory context associated to the entered area. When the area is exited, all subsequent uses of the new operation will allocate memory from the area associated with the enclosing scope.
5. Each real-time thread is associated with a scoped stack containing all the areas that the thread has entered but not yet exited.

Since assignment rules cannot be fully enforced by the compiler, some dangling pointers must be detected at runtime [3]. The more basic approach is to take the advice given in the current edition of the RTSJ specification [6], to scan the scoped area stack associated to the current task, verifying that the scoped area from which the reference is created was pushed in the stack than the area to which the referenced object belongs. This approach requires the introduction of *write barriers* [11]; that is to introduce a code exploring the scoped area stack when creating an assignment. Note that the complexity of an algorithm which explores a stack is *O(n)*, where *n* is the depth of the stack. Since real-time applications require putting boundaries on the time execution of some piece of code, and the depth of the scoped area stack associated with the task of an application are only known at runtime; the overhead introduced by write barriers is unpredictable. In order to fix a maximum boundary or to estimate the average overhead introduced by write barriers, we must limit the number of nested scoped levels that an application can hold [5].

Scoped areas can be nested and each scope can have multiple sub-scopes, in this case the scoped memory hierarchy forms a *tree*. Consider two scoped memory areas, A and B, where the A scoped area is parent of the B area. In such a case, a reference to the A scoped area can be stored in a field of an object allocated in B. But a reference from a field of an object within A to another object allocated in B raises the `IllegalAssignment()` exception. When a thread *enters* a scoped area, all subsequent object allocations come from the entered scoped area. When the thread *exits* the scoped area, and there are no more active threads within the scoped area, the entire memory assigned to the area can be reclaimed along with all objects allocated within it. For the scoped area model behavior, the current edition of the RTSJ specification [6] adds the following rules[1]:

1. The *parent* of a scoped area is the area in which the object representing the scoped area is allocated[2].
2. The *single parent rule* requires that a scope area has exactly zero or one parent.
3. Scope areas that are made current by entering them or passing them as the initial memory area for a new task must satisfy the single parent rule.

In the current RTSJ, when a task or an event handler tries to enter a scoped area S, we must check if the corresponding thread has entered every ancestor of the area S in the scoped area tree. Then, safety of scoped areas requires checking both the set of rules imposed on their entrance and the aforementioned assignment rules. Both tests require algorithms, the cost of which is linear in the number of memory areas that the task can hold. We suppose that the most common RTSJ application uses a scope area to repeatedly perform the same computation in a periodic task. Then, to optimize the RTSJ memory subsystem, we suggest simplifying data structures and algorithms. In order to do that, we propose to change the RTSJ suggested implementation of the parentage relation for scoped areas.

3 The RTSJ Single Parent Rule

The single parent rule guarantees that a parent scope will have a lifetime that is not shorter than of any of its child scopes, which makes safe references from objects in a given scope to objects in an ancestor scope, and forces each scoped area to be at most once in the tree containing all area stacks associated with the tasks that have entered the areas supported by the tree. The single-parent rule also enforces every task that uses an area to have exactly the same scoped area parentage. The implementation of the single-parent rule as suggested by the current RTSJ edition makes the behavior of the application non-deterministic. In the guidelines given to implement the algorithms affecting the scope stack (e.g. the `enter()` method), the single parent rule guarantees

[1] In the former RTSJ edition these rules do not appear.

[2] The suggested implementation of this rule in RTSJ is not correct nor consistent.

that once a thread has entered a set of scoped areas in a given order, any other thread is enforced to enter the set of areas in the same order. Notice that determinism is an important requirement for real-time applications.

Consider tree scoped areas: A, B, and C, and two task τ1 and τ2. Let us suppose that task τ1 has entered areas A and B, and task τ2 has entered areas A and C. If task τ1 tries to enter the area C or task τ2 tries to enter the area B, which violates the single parent rule, the `ScopedCycleException()` is thrown. If, for example, τ1 enters the area C before τ2 tries to enter it, τ2 violates the single parent rule raising the `ScopedCycleException()` exception. But, if τ2 enters the area B before τ2 tries to, then it is τ1 which violates the single parent rule and raises the `ScopedCycleException()` exception.

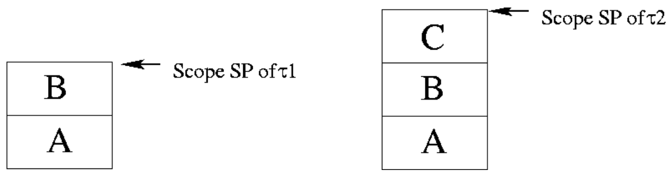
3.1 The Proposed Parentage Relation

In order to maintain the single-parent rule of the current RTSJ edition, we consider that the parent of a scoped area is the area within which the area is created [6], and we add the following rules:

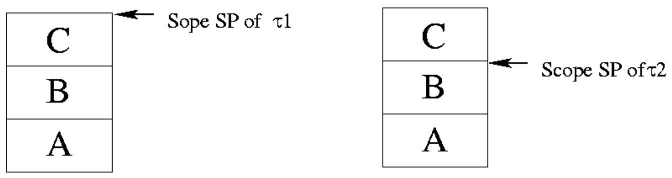
1. The parentage relation of scoped areas implies an scoped area tree structure, where a memory area is added when creating it and deleted when collecting it (instead of when entering and exiting the memory area).
2. Each scoped area must maintain a reference count of the number of scoped areas within which have been created (*child-counter*), in addition to the reference count of the number of threads having it as current area (*task-counter*). The child-counter allows us to maintain alive a scoped area which task-counter values 0 (i.e. it have not yet entered by a task, or it have been entered and exited), but it is the father of the current area of a task.
3. When both reference counters the child-counter and the task-counter for a scoped area reach zero, the scoped area is a candidate to collection.

In this way, as in the current RTSJ edition, we obtain an area tree based on a hierarchy relation. But the parent relation is based on the way that scoped areas are created, instead of the order in which scoped areas have been entered by threads. Consider three scoped areas: A, B, and C, which have been created in the following way: the A area has been created within the heap, the B area has been created within the A area,and the C area has been created within the B area, which gives the following parentage relation: the heap is the parent of A, A is the parent of B, and B is the parent of C. Then, the child-counter for A and B has been incremented to 1, whereas for C it is 0.

Let us further consider the two tasks τ1 and τ2 of our previous example, where we have supposed that task τ1 has entered areas A and B, which increases by 1 the task-counter for A and B. And task τ2 has entered areas A and C, which increases by 1 the task-counter for A and C (see Figure 1.*a*). In this situation, the task-counter for A values 2, whereas for B and C values 1. If task τ1 enters the area C and task τ2 the

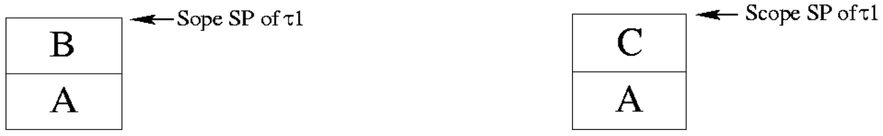


a. τ1 enters B scoped area and τ2 enters C.



b. τ1 enters C scoped area and τ2 enters B.

Fig. 1. The scope stack and the single parent rule.



a. τ1 enters B scoped area. *b. τ1 enters C scoped area.*

Fig. 2. Two diferents stated for the coped stack of task τ1.

area B, at different than those that occur in the suggested implementation of RTSJ [6][3], the single parent rule is not violated. Then, instead of throwing the `ScopedCycleException()`, we have the situation shown in Figure 1.*b*. At this moment, the task-counter for scoped memory areas A, B, and C value 2.

Note that the scoped stack associated to task τ2 includes only the A and B scoped regions. Then, even if the task τ2 has entered the scoped memory C before to enter B, pointers from objects allocated in B to objects allocated in C are dangling pointers, as consequence they are not allowed. We consider another situation: task τ1 enters into scoped area A creates B and C, which increases both the task-counter of A by 1 and its child-counter by 2, whereas both the task-counter and the child-counter for B and C value 0. Then, task τ1 enters into scoped areas B (Figure 2.*a*) and C (Figure 2.*b*), which increases by 1 the task-counter of both B and C. Only references from objects allocated within B or C to objects within A are allowed. Note that it is not possible for task τ1 create a reference from an object within B to an object within C, and vice-versa from an object within B to an object within C, even if task τ1 must exit the area C before to exit the area B. Then, if a task τ2 enters into scoped area C and stays there for a while, task τ1 leaves C and leaves B, the scoped area B can be collected and there are not dangling pointers.

Non-scoped areas (i.e. the heap and immortal areas) are not supported in the scoped tree. Moreover, the heap and immortal areas are considered as the *primordial scope*, which is considered to be the root of the area tree [3]. Notice that, for the heap and immortal memory areas, there is no need to maintain the reference-counters because these areas exist outside the scope of the application.

Then, we propose to change the RTSJ specification, so that scoped memory areas are parented at creation time. This new parentage relation introduces great advantages because *i)* simplifies the semantic of scoped memory as the single parent rule becomes trivially true, *ii)* scope cycle exceptions does not occur, *iii)* each thread requires only one scoped stack, *iv)* and the parentage relation does not change during the scoped memory life.

3.2 Checking the Assignment Rules

We next show how to extend area tree data structures to perform all required checks in constant time. Our approach is inspired in the suggested parentage relation of scoped memory areas. As stated the RTSJ imposed assignment rules, references can always be made from objects in a scoped memory to objects in the heap or immortal memory; the opposite is never allowed. Also the ancestor relation among scoped memory areas is defined by the nesting areas themselves, and this parentage is supported by the area tree. Since area tree changes occur only at determined moments, i.e. when creating or collecting a scoped area, we can apply the technique based on displays that has been presented in [2]. Each scoped area has associated a display containing the type identification codes of its ancestors and its depth in the area tree (see Figure 3).

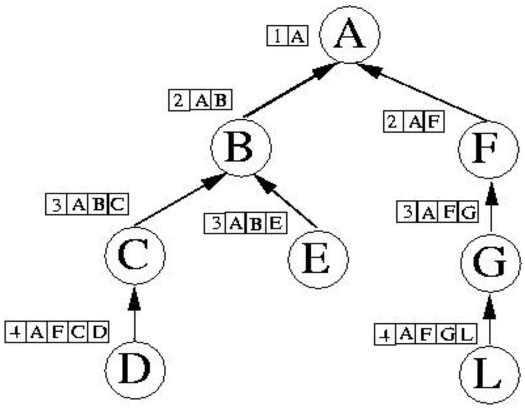


Fig. 3. Display-based area tree structure.

In order to use the display-based technique, we suggest including the following rules in RTSJ, instead of the rule which associates a scope stack to each task:

1. The heap and immortal memory areas are always assigned the minimum depth.
2. When creating a scoped memory area, its corresponding depth is the depth of the father area plus 1.
3. Figure 4 shows the pseudo-code that we must introduce in the execution of each assignment statement (e.g. x.a=y) to perform the assignment checks in constant- time.

```
X = area to which the x object belongs;
Y = area to which the y object belongs;
if( (Y.depth <> 0) and (X.display[Y.depth]<>Y.display[Y.depth])) IllegalAssignment:();
```

Fig. 4. Checking the assignment rules.

3.3 Maintaining the Display Structure

In the current RTSJ specification edition, the `enter()` method can throw the `ScopeCycleException()` whenever entering in a scoped region that would violate the single parent rule. The current RTSJ edition also advises to push/pop the entered region on the scope stack belonging to the current task and to increase/decrease the reference counter of the region when the task enters/exits the `enter()`method (see Figure 5).

```
if entering ma would violate the single parent rule throw ScopedCycleException;
push ma on the scope stack belonging to the current thread;
increase the ma reference count;
execute logic.run method;
decrease the ma reference counter;
pop ma from the scope stack.
```

Fig. 5. Current pseudo-code for ma.enter(logic).

Figure 6 shows the pseudo-code of another operation affecting the scope stack, the construction of a new task. In order to maintain the reference counter collector of scoped regions, we must increase/decrease the reference counter of all regions on the scope stack when creating/destroying the task.

```
cma = curent memory region;
ima = initial memory region;
if cma is heap or immortal
        create a new scope stack containing cma
else
        start a new scope stack containing the entire current scope stack;
for every scoped memory area in the new scope stack increase the reference count;
if ima != current allocation context push ima on new scope stack;
run the new thread with the new scope stack;
when the thread terminates every memory area pushed by the thread will have been popped;
for every scoped memory area in the scope stack decrease the reference count;
free the scope stack.
```

Fig. 6. RTSJ pseudo-code to construct a task.

Whereas in the suggested RTSJ implementation solution, actions are required each time a task is created or enters an area having a *O(n)* complexity, where *n* is the number of nested scoped areas. In our proposed solution, when creating a thread or entering an area the display tree structure is not affected (see Figures 7 and 8). Then, both operations, which associate a scoped memory to a task, become constant execution time. Even if these operations are potentially infrequent, we consider that it is an interesting collateral effect obtaining by changing the parentage relation of scoped memory regions.

```
make  ma the current area;
increase the task reference count of ma;
execute logic.run method;
decrease the task reference count of ma;
restore the previous current area.
```

Fig. 7. The proposed enter() method using displays.

```
ima = initial memory area;
make  ima the current area;
increase the task reference count of ima;
run the new thread ;
when the thread terminates decrease the task reference count of ima
```

Fig. 8. Constructing a task by using displays.

The current RTSJ edition also presents another method allowing a task to change the allocation context; the `executeInArea()` method, which checks the current scope stack in order to find the area to which the message associated with the method is sent. Since these methods require an exploration of the stack, they have an *O(n)* complexity. Notice that in our proposed solution entering an area older than the current one that is in the same branch of the area tree (i.e. in the same scope stack), has the same consequences as the `executeInArea()` method. Therefore, this method is not strictly necessary, and actually it does not appear in the former edition of the RTSJ specification.

3.4 Estimating the Write Barrier Overhead

We have modified the KVM [7] to implement three types of memory areas: *(i)* the heap that is collected by the KVM GC, *(ii)* immortal that is never collected and can not be nested, and *(iii)* scoped that have limited live-time and can be nested. To obtain the introduced write barrier overhead, two measures are combined: the number of events, and the cost of the event. We use an artificial collector benchmark which is an adaptation made by Hans Boehm from the John Ellis and Kodak benchmark. This benchmark executes $262*10^6$ bytecodes, where $15*10^6$ performs a store operation (i.e., aastore: $1*10^6$, putfield: $6*10^6$, putfield_fast: $7*10^6$, putstatic: $19*10^6$, and putstatic_fast: 0). That is, the 5% of executed bytecodes perform a write barrier test, as already obtained with SPECjvm98 in [8]. The write barrier cost is proportional to the number of executed evaluations. With this implementation, the average write barrier cost is only 1.6%. Note that this low overhead is not realistic because the KVM is a very small and very slow JVM.

4 Related Works

The enforcement of the RTSJ assignment rules includes the possibility of static analysis of the application logic [6]. Escape analysis techniques could be used in order to remove run-time checks. But the dynamic issues that Java presents, requires for some cases to check the assignment rules at run-time. However, static and dynamic techniques can be combined to provide more robustness and predictability of RTSJ applications. The idea of using both write barrier and a stack of scoped areas ordered by life-times to detect illegal inter-area assignments was first introduced in [4]. The most common approach to implement read/write barriers is by inline code, consisting in generating the instructions executing barrier events for every load/store operation. This solution requires compiler cooperation and presents a serious drawback because it increases the size of the application object code [9]. This approach is taken in [1] where the implementation uses five runtime heap checks (e.g. CALL, METHOD, NATIVECALL, READ, and WRITE). Alternatively, our solution instruments the bytecode interpreter, avoiding space problems, but this still requires a complementary solution to handle native code. The display-based technique was firstly used to support RTSJ scoped area in [2]. The main difference between both techniques is that encoding of the type hierarchy in [5] is known at compile time, whereas in [2] the area tree changes at runtime. This technique has been extended to perform memory access checks in constant-time. The main contribution of our approach is to avoid the single parent checks by changing the parentage relation of scoped area within the area tree, which ensures all algorithms managing scoped areas to be executed in constant-time.

5 Conclusions

To enforce the RTSJ imposed rules, a compliant JVM must check both the single parent rule on every attempt to enter a scoped memory area, and the assignment rules on every attempt to create a reference between objects belonging to different memory areas. Since objects references occur frequently, it is important to implement checks for assignment rules efficiently and predictably. The parentage relation of areas is based on the way they are created/collected, instead of the way they are entered/exited by tasks such as the RTSJ suggests. Then, we avoid checking on every attempt to enter a scoped memory area. Also, changes on the proposed stack based structure are less frequent, which allows us use more simple algorithms. The scope stack can be coded as a display, which allows us to use subtype test based techniques making the enforcement of memory references time-predictable, and does not depend of the nested level of the area to which the two objects of the memory reference belong.

In order to do the implementation of required algorithms more efficient, every scoped area has two reference counters and a scoped stack associated to it, which allows us a more efficient management of areas, making it time predictable. Note that by collecting areas, problems associated with reference-counting collectors are solved: the space and time to maintain two reference-counts per scoped area is minimal, and there are no cyclic scoped area references. Note that the introduction of this change in the parentage relation simplifies the complex semantics for scoped memory region adopted by RTSJ.

References

1. W.S. Beebe and M. Rinard. “An Implementation of Scoped Memory for Real-Time Java”. In Proc of 1st International Workshop of Embedded Software (EMSOFT), 2001.
2. A. Corsaro and R.K. Cytron. “Efficient Reference Checks for Real-time Java. "ACM SIGPLAN Conference On Languages, Compilers, and Tools for Embedded Systems", LCTES 2003.
3. P.C. Dibble. “Real-Time Java Platform Programming”. Prentice Hall 2002.
4. M.T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J.P. Lesot, and F. Parain. "Area-based Memory Management for Real-time Java". In Proc. of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC). IEEE 2001.
5. M.T. Higuera and, V. Issarny “Analyzing the Performance of Memory Management in RTSJ”. In Proc. of the 5th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC). IEEE 2002.
6. The Real-Time for Java Expert Group. "Real-Time Specification for Java". RTJEG 2002. http://www.rtj.org
7. Sun Microsystems. “KVM Technical Specification". Technical Report. Java Community Process, May 2000. http://java.sun.com.
8. Standard Performance Evaluation Coorporation: SPEC Java Virtual Machine Benchmark Suite. http://www.spec.org/osg/jvm98, 1998.
9. Zorn B. “Barrier Methods for Garbage Collection”. Technical Report CU.CS. Department of Computer Science. University of Colorado at Boulder. http://www.cs.colorado.edu. November 1990.